Appendix 1

SMART
Smart Multi-Array Routing Table

Yoichi Hariguchi, Tom Herbert
MAYAN Networks Corp.
{yoichi,therbert}@mayannetworks.com

Abstract— **It is known that the routing table based on multibit trie, in other words, multiple levels of arrays (hereafter let us call it MART, Multi-Array Routing Table) has a very low and deterministic search cost of modestly higher memory consumption compared to other routing table approaches. The search cost of MART is typically 2 to 4 routing table memory accesses for IPv4. MART has the additional benefits that it is easy to implement in hardware and its determinism allows for pipelining route lookups in hardware. The primary drawback of MART is that update operations, namely adding and deleting routes, are highly costly relative to the route lookup operation. Our extension to MART, called "Smart Multi-Array Routing Table" or just SMART, addresses this issue. SMART not only has similar low cost and determinism of route lookups but also provides low cost route update operations, which always have lower than 256 routing table memory accesses regardless of both the number of routes in the routing table and the prefix length for IPv4.**

Keywords— **Longest-Matching, Route Lookup, Internet Protocol**

I. Introduction

THE size of the Internet routing table is growing rapidly [1] even after the introduction of CIDR (Classless Inter-Domain Routing) [2]. The number of routes at MAE-EAST is about 50,000 [3] as of this writing. In addition, the routing instability [4] is becoming a serious problem. This problem is also called 'route flap'. The route flap often causes deleting and adding the entire set of BGP (Border Gateway Protocol) routes. It is important to enhance not only route lookup performance but also route update performance because slow route update may cause a route flap storm.

It is known that MART has a very low and deterministic search cost. For example, the route lookup cost of a MART implementation by Pankaj Gupta, Steven Lin, and Nick McKeown [5] is 2 memory accesses in the worst case. However, their MART implementation has a serious drawback, which is its highly expensive update cost. It needs 32M routing table memory accesses (16M reads and 16M writes) to add a single route in the worst case. Another MART implementation by Srinivasan and Varghese [6] has much better worst case update cost, but it still does not solve a problem of MART at route deletion. Actually, neither paper discusses the problem of route deletion. In this paper, we first discuss the problem of MART at route deletion and then introduce a new routing table design called SMART (Smart Multi-Array Routing Table) that solves the problem. SMART is an extension of MART and its route update cost is always smaller than 256 routing table memory accesses (128 writes and 127 reads).

II. Previous Work

FIGURE 1 shows an example of simple MART.

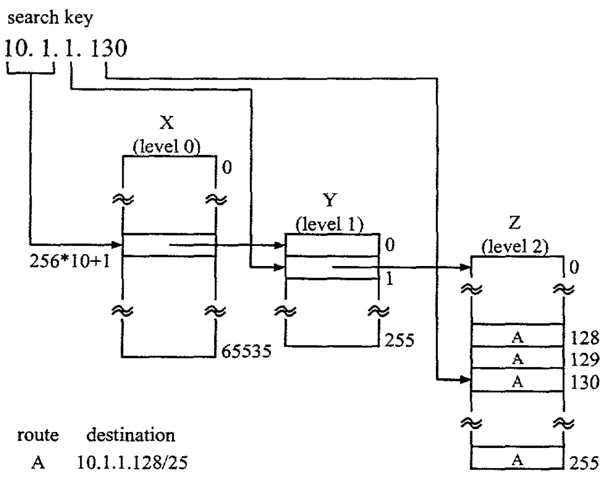


Fig. 1. Simple Multi-Array Routing Table

The MART in Figure 1 consists of 3 levels of arrays. The level 0 array X is indexed by the most significant 16 bits of the search key IPv4 address and has 64K elements. The level 1 array Y is indexed by bit 8-15 of the search key IP address and has 256 elements. The level 2 array Z is indexed by the least significant 8 bits of the search key IPv4 address and has 256 elements. In other words, each IP address can be mapped to one element of the arrays of level 0, level 1, or level 2. Hereafter we will focus on this 3-level MART and SMART. The same theory can be

```
For i = 2,560 (10.0) to 2,815 (10.255)
  If X[i] is connected to a level 1 array then
    Level 1 array Y = X[i]
    For j = 0 to 255
      If Y[j] is connected to a level 2 array then
        Level 2 array Z = Y[j]
        For k = 0 to 255
          If Z[k] == NULL or prefix length of
              the route pointed to by Z[k] < 8 then
            Z[k] = B
          Endif
        Endfor
      Else if Y[j] == NULL or prefix length of
              the route pointed to by Y[j] < 8 then
        Y[j] = B
      Endif
    Endfor
  Else if X[i] == NULL or prefix length of
          the route pointed to by X[i] < 8 then
    X[i] = B
  Endif
Endfor
```

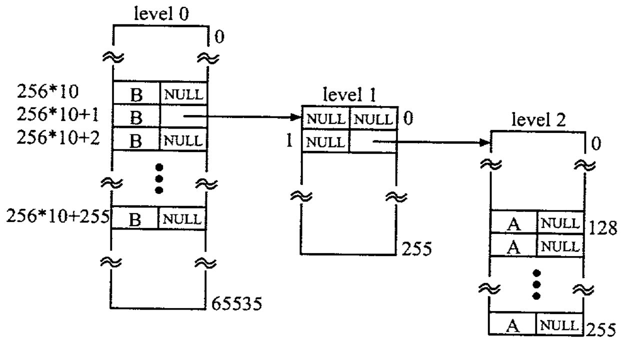
Fig. 2. Adding Route 10/8 to MART in Figure 1

applied to any variation.

When a route is added, all of the array elements of the routing table that correspond to the destination IP prefix of the route are set to point to the added route. For example, the destination IP prefix of route A in Figure 1 is 10.1.1.128/25. That is why the elements from 128 to 255 in the level 2 array corresponding to 10.1.1 have pointers to route A. The MART in Figure 1 always finishes a lookup within 3 routing table memory accesses.

Assume a new route 'route B' whose destination IP prefix is 10/8 is to be inserted to the MART in Figure 1. Figure 2 shows a pseudo code to add route B to the routing table in Figure 1. Figure 2 indicates that it takes 16M (256×256×256) routing table memory reads and 16M routing table memory writes to add one entry in the worst case.

An idea called 'Controlled Prefix Expansion' in a paper by Srinivasan and Varghese [6] reduces the route update cost of MART. Hereafter, let us call their MART implementation MART-CPE. MART-CPE has two pointers per element. The one pointer (say *pCPE*) points to the longest-matching route associated with the element. The other pointer (say *pNextLevel*) points to the next level array if it exists. The MART in Figure 1 and 2 spreads a pointer to the longest-matching route all over the routing table. In contrast, MART-CPE spreads a pointer to the longest-matching route only within an array. Figure 3 shows the MART-CPE routing table that has route A and route B. The reason MART-CPE does not have to spread a pointer all over the routing table is that MART-CPE prepares a variable called *BMP* and updates *BMP* with the value of *pCPE* each time MART-CPE visits a deeper level array at route lookup. The route addition cost of MART-CPE is 256 memory accesses in the worst case.



route	destination
A	10.1.1.128/25
B	10/8

Fig. 3. MART with Controlled Prefix Expansion

III. Enhancement of MART in Deletion

Assume now route A is to be removed. In the case of the routing table in Figure 1, the pointers of element 128 to 255 of the level 2 array must be replaced with the new longest-matching route after route A is removed, which is route B. In the case of the routing table in Figure 3, *pCPE* of element 128 to 255 of the level 2 array must be replaced with the new longest-matching route after route A is removed, which is NULL. Neither the paper by Gupta, Lin, and McKeown [5] nor the paper by Srinivasan and Varghese [6] discusses how to find the new longest-matching route to be replaced with the route to be deleted. The simplest way is to backtrack through the array elements (and arrays in the case of MART). This process is expensive. The simple backtrack takes 254 memory accesses in the worst case even in MART-CPE. There is a better way to backtrack through the array elements. Assume there is a route whose IP prefix is 10.1.1.252/30 in the MART-CPE in Figure 3. In this case, there are 6 possible less specific routes than 10.1.1.252/30 in the array, which are 10.1.1.248/29, 10.1.1.240/28, 10.1.1.224/27, 10.1.1.192/26, 10.1.1.128/25, and 10.1.1/24. That is why it is enough to check 6 array elements corresponding to the above routes instead of checking all of 254 array elements. However, it still needs to check up to 6 array elements. In addition, it costs to calculate these indices of the array element.

SMART solves this backtracking problem of MART at route deletion. SMART always finds the route to be replaced with the route to be deleted at one memory access without any backtracking.

IV. SMART

FIGURE 4 shows the array element structure of SMART.

pBlkDef	pLongestMatch	pNextLevel

Fig. 4. SMART Array Element

Each array element of SMART has 3 pointers. Suppose there is an element whose index is i in array X. Let us call the element 'element i of Array X', or simply 'element i' unless it is necessary to specify an array. Let us also define the term 'base element' to refer to the first array element in the routing table at which the destination IP prefix of a route fully matches. For example, the base element of route A becomes element 128 of a level 2 array.

Pointer *pLongestMatch* of element i points to the first element of a list of routes whose base element is i. The list is kept sorted in the descending order of prefix length of the destination IP prefix of the routes (hereafter we call the prefix length of the destination IP prefix of a route 'prefix length of a route'). This is because it is possible that multiple routes can have the same base element. We discuss this issue in Section IV-E. Pointer *pNextLevel* points to the next level array if it exists. Otherwise *pNextLevel* is set to NULL. The value of *pBlkDef* of element i is equal to the value of *pLongestMatch* of another element (say element j) that matches the following conditions:

1. $0 \leq j < i$
2. the prefix length of the route pointed to by *X[j].pLongestMatch* is shorter than the prefix length of the route pointed to by *X[i].pLongestMatch*
3. the prefix length of the route pointed to by *X[j].pLongestMatch* is larger than the prefix length of the routes pointed to by *X[k].pLongestMatch* $(0 \leq k < j)$

We call the route pointed to by *pBlkDef* 'block default route'. When the value of *pLongestMatch* of element i is NULL, the block default route of element i becomes the longest-matching route associated with element i. It means that a block default route points to the second longest-matching route of the associated element. This characteristic solves the backtracking problem at the route deletion. We discuss this in section IV-C. When both *pLongestMatch* and *pBlkDef* are NULL, the longest-matching route associated with element i is pointed to by *pBlkDef* of the element in the previous level array whose *pNextLevel* points to array X.

A. Route Addition

Assume there are no routes in the routing table, then a route whose destination IP prefix is 10.1.4/22 (say route C) is added. The route addition process is:

1. A level 1 array (let us call it array *X*) is allocated and cleared.
2. *L0[10×256+1].pNextLevel* = *X*. Here and hereafter, L0 is the level 0 array.
3. *X[4].pLongestMatch* = C
4. *X[5..7].pBlkDef* are set to C

Figure 5 shows level 1 array *X* after route C is added. Note that the block default route of element 4 (the route pointed to by *X[4].pBlkDef*) is not route C. *X[4].pBlkDef* must be one of the following 3 values by definition:

1. *X[0].pLongestMatch* unless *X[0].pLongestMatch* is NULL
2. NULL if *X[0].pLongestMatch* is NULL.

In this case, *X[4].pBlkDef* is set to NULL since both *X[0].pLongestMatch* and *X[2].pLongestMatch* are NULL. The routes pointed to by *X[1].pLongestMatch* and *X[3].pLongestMatch* cannot be the block default route of X[4] since their prefix length length must be 24.

Element	Contents		
	pBlkDef	*pLongestMatch*	*pNextLevel*
0	NULL	NULL	NULL
1	NULL	NULL	NULL
2	NULL	NULL	NULL
3	NULL	NULL	NULL
4	NULL	C	NULL
5	C	NULL	NULL
6	C	NULL	NULL
7	C	NULL	NULL
8	NULL	NULL	NULL
⋮	⋮	⋮	⋮
255	NULL	NULL	NULL

C: 10.1.4/22

Fig. 5. Level 1 Array X After Adding Route C (10.1.4/22)

Assume now a route whose destination IP prefix is 10.1/20 (let us call it route D) is added. The route addition process in this case is as follows:

1. *X[0].pLongestMatch* = D

2. *X[1..15].pBlkDef* are set to D if their value is NULL or the prefix length of the destination IP prefix pointed to by each of them is shorter than 20.

The block default routes of element 5 to 7 do not change since they are pointing to route C and the prefix length of route C is 22. Figure 6 shows the level 1 array after route D is added.

Element	Contents		
	pBlkDef	*pLongestMatch*	*pNextLevel*
0	NULL	D	NULL
1	D	NULL	NULL
2	D	NULL	NULL
3	D	NULL	NULL
4	D	C	NULL
5	C	NULL	NULL
6	C	NULL	NULL
7	C	NULL	NULL
8	D	NULL	NULL
⋮	⋮	⋮	⋮
15	D	NULL	NULL
16	NULL	NULL	NULL
⋮	⋮	⋮	⋮
255	NULL	NULL	NULL

C: 10.1.4/22
D: 10.1/20

Fig. 6. Level 1 Array X After Adding Route D (10.1/20)

Assume now a route whose destination IP prefix is 10/8 (route B) is added. In this case, the route addition process is as follows:

1. *L0[10×256].pLongestMatch* = B.
2. *L0[10×256+1 .. 10×256+255].pBlkDef* = B since their value is NULL.

Note that neither level 1 nor level 2 array is accessed. Figure 7 shows the whole SMART after route B is added.

Figure 8 shows a pseudo code of some functions used for routing table operations for the 3-level SMART. It is necessary to change only these functions to support arbitrary-level SMART.

Figure 9 shows a generic route addition pseudo code for SMART. This pseudo code can be used for arbitrary-level SMART. The index of the last array element in which the new route is stored as the block default route is equal to the index of the last array element of the associated IP address belonging to the IP prefix of the new route. Thus the maximum number of routing table memory access at route addition is 511 (256 writes and 255 read).

The simplest implementation visits all the array elements associated with the new route and checks the route pointed to by *pBlkDef*, but it is not necessary. When a route is stored in *pLongestMatch* of element *i* of array *X*, in other words, *X[i].pLongestMatch* is not NULL, there is

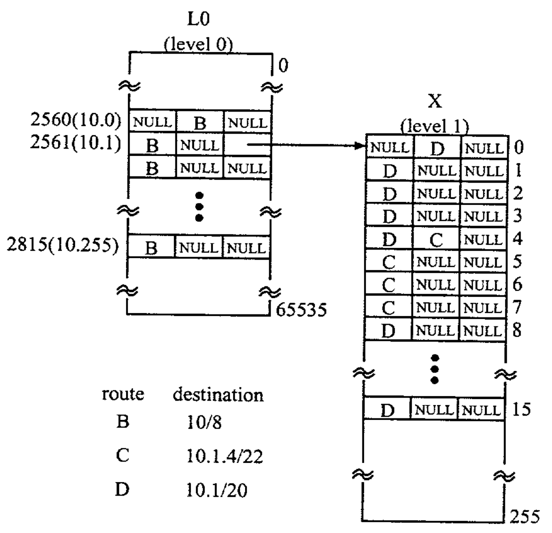


Fig. 7. 3 Routes in SMART

no need to check the value of *pBlkDef* of the elements associated with the route pointed to by *X[i].pLongestMatch* since these routes are always more specific than the new routes. It means that the number of routing table memory accesses can be reduced when routes are added in the descendent order of prefix length. We discuss this issue in Section VII.

```
/* getLevel(plen) returns the level of an array from the prefix
length plen */
    getLevel(plen)
      If plen ≤ 16 then
        return 0
      Endif
      If plen ≤ 24 then
        return 1
      Endif
      return 2

/* getIndex() returns the index of an array associated with the fol-
lowing parameters: IP address ipa, prefix length plen */
    getIndex(ipa, plen)
      If plen ≤ 16 then
        return ipa >> 16
      Endif
      If plen ≤ 24 then
        return (ipa >> 8) & 0xff
      Endif
      return ipa & 0xff
```

```
/* getIndexFromLevel() returns the index of an array associated
with the following parameters: IP address ipa, array level level */
    getIndexFromLevel(ipa, level)
        If level == 0 then
            return ipa >> 16
        Endif
        If level == 1 then
            return (ipa >> 8) & 0xff
        Endif
        return ipa & 0xff
```

```
/* getNscan() returns the number of elements whose block default
routes need to be updated from the prefix length plen */
    getNscan(plen)
        return $(1 << ((16 + getLevel(plen) \times 8) - plen)) - 1$
```

```
/* getMaxLevel() returns the maximum level of array in SMART
*/
    getMaxLevel()
        return 2
```

```
/* getExactMatch() returns pointer to the exact matching route in
the linked list starting X[index].pLongestMatch */
    getExactMatch(ipa, plen, X, index)
        If X[index].pLongestMatch == NULL then
            return NULL
        Foreach element in linked list
                    starting from X[index].pLongestMatch
            If element→ipa == ipa &&
                        element→plen == plen then
                return element
            Endif
        Endforeach
        return NULL
```

```
/* get2ndBest() returns pointer to the 2nd best matching route as-
sociated with X[index] */
    get2ndBest(X, index, pRoute)
        If pRoute is the last element of the linked list
                    starting from X[index].pLongestMatch
            return X[index].pBlkDef
        Endif
        return the next element of pRoute
```

Fig. 8. Functions Used for Addition and Deletion

```
addRoute(ipa, plen)
/* ipa: IP address, plen: prefix length */

    /* Allocate new array(s) if necessary */
    array X = L0     /* level 0 array */
    i = 0     /* i indicates level */
    While i < getLevel(plen)
        index = getIndexFromLevel(ipa, i)
        If X[index].pNextLevel == NULL then
            Array tmp = a newly allocated array
            Clear tmp
            X[index].pNextLevel = tmp
        Endif
        X = X[index].pNextLevel /* get next level array */
        i = i + 1
    Endwhile

    /* Update elements of array X */
    begin = getIndex(ipa, plen)
    nScan = getNscan(plen)
    Insert the new route to the list starting from
        X[begin].pLongestMatch with the prefix lengths
            in the list kept in the descending order
    i = begin + 1     /* i now indicates array index */
    While nScan > 0
        If X[i].pBlkDef == NULL or
                plen > prefix length of the route
                    pointed to by X[i].pBlkDef then
            X[i].pBlkDef = pointer to the new route
        Endif
        If X[i].pLongestMatch == NULL then
            nSkip = 1
        Else
            nSkip = getIndex(~X[i].
                        pLongestMatch→mask, level) + 1
        Endif
        i = i + nSkip
        nScan = nScan - nSkip
    Endwhile
```

Fig. 9. SMART Route Addition Algorithm

B. Search

Figure 10 shows a route lookup pseudo code for SMART. The pseudo code in Figure 10 can be used for arbitrary-level SMART. The following is an example tosearch the routing table shown in Figure 7 for IP address 10.1.17.1:

1. Local variable *pBest* is initialized to NULL.
2. Element 2,561 (10.1) of the level 0 array (*L0*) is accessed. *pBest* is set to B.
3. Element 17 of the level 1 array is accessed. The value of *pBest* does not change since both *pLongestMatch* and *pBlkDef* of element 17 are NULL.
4. *pBest*, which is pointing to route B, is returned since *pNextLevel* of element 17 is NULL.

Each time an element of a deeper level array is visited, it is necessary to update local variable *pBest* with either *pLongestMatch* or *pBlkDef* of the visited element as described in Figure 10. This is because the route pointed to by one of these pointers becomes the longest-matching route when all 3 pointers (*pLongestMatch*, *pBlkDef*, and *pNextLevel*) of the element in the next level array are NULL. It means that the search cost of SMART is 3 times as expensive as the MART in Figure 1 in the worst case since SMART requires 3 memory reads per element visit. In contrast, MART needs one memory read per element visit. The difference of the search cost between the two is however very small in reality. This is because:

1. If one pointer is in the CPU cache or the cache of search hardware, other 2 pointers are also in the cache in the most of cases since these 3 pointers are connected. That is why the possibilities are: 1) all 3 pointers are in the cache, 2) none of 3 pointers is in the cache. In case 1, the cost of 2 more memory accesses is the same as that of 2 more register instructions. In case 2, the cost of 2 more memory accesses is negligible since the cost of loading 3 pointers into the cache is more than 10 times as expensive as that of 2 more cache accesses. That is why the cost of 2 more memory accesses is at the worst 2 more register instructions.
2. It is possible to make it parallel to update *pBest* and check *X[index].pNextLevel* in hardware.
3. There is a way to reduce the number of members per element from 3 to 2. We discuss this optimization of the basic SMART in Section V.

We will show the simulation result of the search performance for MART and SMART in section VII.

C. Route Deletion

When a route (let us call it route Q) is deleted, it is necessary to update all the array elements that have a pointer to route Q throughout the routing table in MART. In contrast, MART-CPE and SMART require to update only the elements in the same array in which pointers to route Q were stored.

The update consists of two parts. The one is to find the route to be replaced (let us call it route R) with route Q. Route R must be the second longest-matching route of the destination IP address of route Q. The other is to replace the pointer value in all the necessary array elements as described above. As we saw in section II, it is expensive to find route R in both MART and MART-CPE since it is necessary to backtrack through the array elements (and arrays in the case of MART). SMART can always find route R with one memory access. Suppose a pointer to route Q is stored in *X[i].pLongestMatch* (element *i* of array X). Route R is always stored in *X[i].pBlkDef* because a block default route always points to the second longest-matching route of the associated element.

Figure 11 shows a route deletion pseudo code for SMART. This pseudo code can be used for arbitrary-level SMART. The maximum number of routing table memory access in Figure 11 is 511 (256 writes and 255 read).

Same as the case of route addition, it is not necessary to visit all the array elements associated with the route to be deleted when *X[i].pLongestMatch* is not NULL. It means that the number of routing table memory accesses can be reduced when routes are deleted in the ascendent order of prefix length. We discuss this issue in Section VII.

```
lookupRoute(ipa) /* ipa: IP address */
  pBest = NULL
  array X = L0     /* level 0 array */
  For i = 0 to getMaxLevel()
    index = getIndexFromLevel(ipa, i)
    If X[index].pLongestMatch ≠ NULL then
      pBest = X[index].pLongestMatch
    Elseif X[index].pBlkDef ≠ NULL then
      pBest = X[index].pBlkDef
    Endif
    If X[index].pNextLevel == NULL then
      break
    Endif
    X = X[index].pNextLevel
  Endfor
  return pBest
```

Fig. 10. SMART Search Algorithm

```
delRoute(ipa, plen)
/* ipa: IP address, plen: prefix length */

  /* Find the element to be deleted */
  array X = L0      /* level 0 array */
  For i = 0 to getLevel(plen)
    index = getIndexFromLevel(ipa, i)
    If X[index].pNextLevel == NULL then
      break
    Endif
    save[i] = X     /* save X for freeing */
    X = X[index].pNextLevel /* get next level array */
  Endfor
  pRoute = getExactMatch(ipa, plen, X, index)
  If pRoute == NULL then
    return      /* no such route */
  Endif

  /* Update elements in array X */
  pDef = get2ndBest(X, index, pRoute)
  begin = index
  nScan = getNscan(plen)
  index = begin + 1
  While nScan > 0
    If X[index].pBlkDef ==
                  X[begin].pLongestMatch then
      X[index].pBlkDef = pDef
    Endif
    If X[index].pLongestMatch == NULL then
      nSkip = 1
    Else
      nSkip = getIndex(~X[index].
                  pLongestMatch→mask, level) + 1
    Endif
    index = index + nSkip
    nScan = nScan - nSkip
  Endwhile
  X[begin].pLongestMatch = NULL

  /* Free array(s) if necessary */
  While i > 0
    If all of pNextLevel in save[i] are NULL
        and all of pLongestMatch in save[i]
                          are NULL then
      free save[i]
    Endif
    i = i − 1
  Endwhile
```

Fig. 11. SMART Route Deletion Algorithm

D. Table Default Route

We saw it takes 511 routing table memory accesses to update a route in the worst case. The maximum number of routing table memory access can be reduced to 255, which is the second largest number of routing table memory access at route update. Note that the worst case happens only when the base element of the route is 0 and the return value of *getNscan(plen)* in Figure 8 is equal to the size of the array. Here, *plen* is the prefix length of the route. Thus we can store the block default route to a different place in an array instead of storing it to all the array elements. We call this single route 'table default route'. The table default route is checked when a pointer to the block default route is NULL at search time. The table default route reduces route the update cost 50%. The drawback, however, is that the table default route method requires an extra check when a pointer to the block default route is NULL.

E. Overlapping Routes

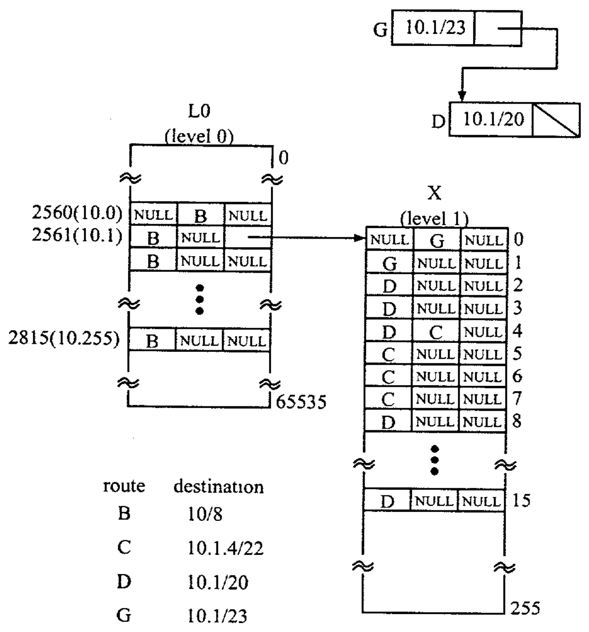
Assume a route 'route G' whose destination IP prefix is 10.1/23 is added to the routing table in Figure 7. Route G is supposed to be stored in *X[0].pLongestMatch* in Figure 7 but route E is already stored in *X[0].pLongestMatch*. This happens when multiple routes have the same destination IP address and each of their prefix length is different. Let us call these routes 'overlapping routes'.

Our SMART implementation makes a linked list of the route entries of overlapping routes. In addition, the overlapping route entries are linked in the descending order of prefix length so that no extra search cost is introduced since the first list element always keeps a pointer to the longest-matching route. Figure 12 shows the routing table after route G is added. It is important to notice that the block default route of route G is not the route pointed to by *X[0].pBlkDef* but route D. The following shows how to find the block default route for overlapping routes:

```
If the route has the shortest prefix length
            among the routes in the linked list then
  block default route is stored in the array element
Else
  block default route is stored
                  in the next linked list element
Endif
```

V. Optimization

We described the basic SMART routing table operations in the previous section. This section discusses an optimization to basic SMART, particularly how to reduce the memory usage and how to increase the search



route	destination
B	10/8
C	10.1.4/22
D	10.1/20
G	10.1/23

Fig. 12. Overlapping Routes (D and G)

performance.

A basic SMART array element has 3 members (Figure 4). This structure simplifies the routing table operations, but requires a lot of memory. We can consolidate *pLongestMatch* and *pNextLevel* as shown in Figure 13 by using the least significant 2 bits of *pBlkDef* as the union identifier of *pNextLevel*.

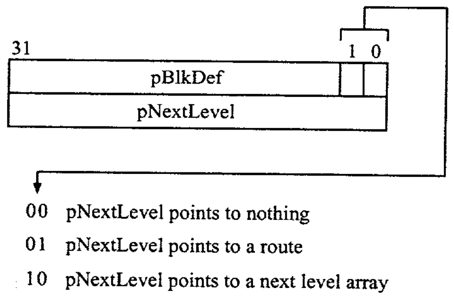
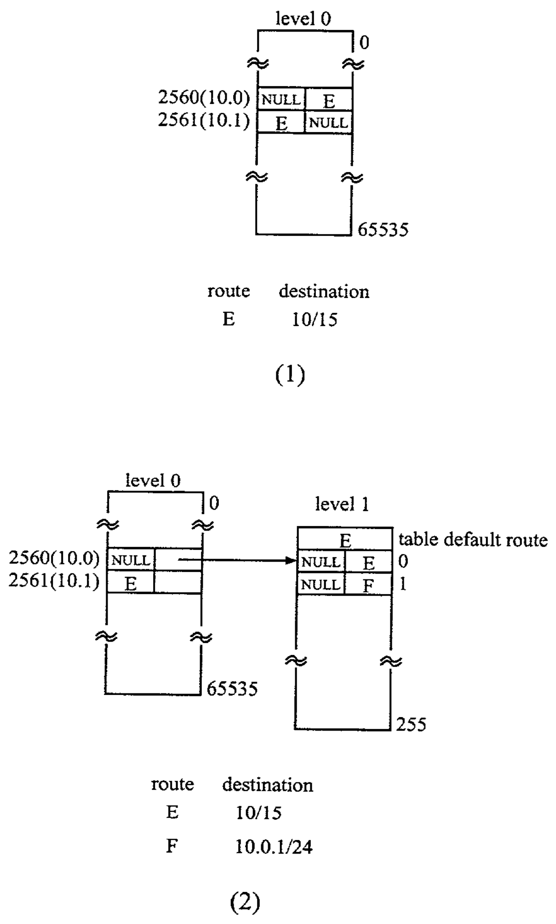

31
1 0
pBlkDef
pNextLevel
00 pNextLevel points to nothing
01 pNextLevel points to a route
10 pNextLevel points to a next level array

Fig. 13. Optimized Smart Array Element

This enhancement saves 33% of memory per element. It also reduces the worst case search cost from 3 to 2 routing table memory accesses per element visit. However, it causes the following 'moving routes' problem. Assume the situation in Figure 14-1. The SMART in Figure 14-1 has only one route, which is route E whose destination IP prefix is 10/15. Assume now a new route 'route F' whose destination IP prefix is 10.0.1/24 is to be added to the table. The route adding function has to:

1. move route E from the level 0 array to the level 1 array as shown Figure 14-2
2. update the block default routes or the table default route of the level 1 array.

This means the block default routes or the table default route may spread to multiple arrays. The table default route is effective in this case because a route to be moved always becomes the table default route in the next level array so that it does not affect the worst case route update cost. The route deletion function also has to handle the spread block default routes or the table default route properly.



route	destination
E	10/15

route	destination
E	10/15
F	10.0.1/24

Fig. 14. Moving a Route

VI. Cost Comparison

Table I shows the maximum number of routing table memory accesses (MAX-RTA) of the routing table operations among MART, MART-CPE, and SMART. SMART-TD is an implementation of SMART that uses the table default route method. Let us use W for the length of an address (e.g., 32 for IPv4, 64 for IPv6 [7]), N for the number of sequences (levels), and S for the maximum number of elements in an array.

Algorithm	Search	Addition	Deletion
MART (original)	N	$2S^N$	$NS + 2S^N$
MART (enhanced)	N	$2S^N$	$NlogS + 2S^N$
MART-CPE (original)	$2N$	$2S$	$S + 2S$
MART-CPE (enhanced)	$2N$	$2S$	$logS + 2S$
SMART	$2N$	$2S$	$1 + 2S$
SMART-TD	$2N$	S	$1 + S$

TABLE I
MAXIMUM NUMBER OF ROUTING TABLE MEMORY ACCESSES (MAX-RTA)

In Deletion, the left hand side number shows the maximum number of routing table memory accesses to search for the second longest matching route with which the route to be deleted is replaced; the right hand side number shows the maximum number of routing table memory accesses to update a subtable.

A. Search

MAX-RTA of MART is N because it requires single memory access per array visit.

MAX-RTA of MART-CPE is $2N$ because it requires 2 memory accesses (reading *pCPE* and *pNextLevel*) per array visit.

MAX-RTA of SMART depends on the implementations. It is $3N$ in the case of 3 members per element, and $2N$ in the case of 2 members per element. However, there is no difference between MART and SMART in reality. We will show this in Section VII.

B. Addition

MAX-RTA of MART is $2S^N$ because it spreads a pointer all over the routing table. it is necessary to multiply 2 because MART has to read the pointer of a route first, then it writes a different pointer if necessary.

MAX-RTA of MART-CPE is $2S$ because it spreads a pointer within an array. It becomes S when the table default method is used.

C. Deletion

The paper [5] does not describe how to find the second longest matching route at all. MAX-RTA of the simplest MART is $NS + 2S^N$. It becomes $Nlog_2S + 2S^N$ in our enhanced MART.

The paper [6] does not describe how to find the second longest matching route at all. MAX-RTA of the simplest MART-CPE is $S + 2S$. It becomes $log_2S + 2S$ in our enhanced MART-CPE.

MAX-RTA of SMART is $1+2S$. It becomes $1+S$ when the table default method is used.

VII. SIMULATION RESULTS

WE simulated all 3 routing table operations, which are addition, deletion, and lookup, for the BSD radix implementation [8], a MART implementation, and a SMART implementation that has 2 members per array element. The reason we used our sample MART implementation is that we could not find any source code of MART. We used MAE-EAST routing table [3] on Aug. 17, 1999 as the source of BGP routes. This routing table has 42,366 routes and 95 routes have the prefix length longer than 24. We also created 2,000 random routes whose prefix length is longer than 24 and added them to the routing table as the source of IGP (Interior Gateway Protocol) routes. This is because MAE-EAST routing table does not have IGP routes whose prefix lengths are usually longer than 24. It is important to include the IGP routes whose prefix length is longer than 24 because it significantly increases the number of both level 1 array and level 2 array (see Table V). One should not ignore IGP routes at simulation because it is common that ISP (Internet Service Provider) routers have both BGP and IGP routes in their routing tables in the real world, and again, the prefix lengths of most of the IGP routes are longer than 24. Table II shows the prefix length distribution of the routing table used for the simulation.

prefix length	number of routes	prefix length	number of routes	prefix length	number of routes
8	20	17	496	25	435
9	3	18	1081	26	438
10	3	19	3285	27	407
11	9	20	1723	28	406
12	21	21	2082	29	406
13	42	22	2790	30	3
14	110	23	3582		
15	184	24	21971		
16	4869				

Total Number of Routes: 44,366

TABLE II
PREFIX LENGTH DISTRIBUTION

The simulation process:

1. randomly adds 44,366 routes to the routing table

2. looks up 100,000 random IP addresses in the routing table
3. randomly deletes all 44,366 routes from the routing table
4. repeats 1 to 3 for 100 times.

The simulation adds, looks up, and deletes routes randomly so that we can avoid any order dependent effects. The simulation is performed on a machine that has the following specifications:

- CPU: AMD Athlon 600MHz
- Memory: 128MB
- OS: Linux 2.2.14
- Compiler: egcs-2.91.66

Table III shows the simulation result of the tree routing table implementations.

	search (μs/route)	add (μs/route)	delete (μs/route)	memory use (MB)
BSD Radix	2.00	4.06	3.61	5.54
MART	0.20	2.03	3.83	9.98
MART-CPE	0.20	1.13	2.03	17.42
SMART	0.20	1.35	1.35	17.42
SMART-TD	0.20	1.13	1.13	17.42

- The values in μs are the average of 44,366 random addition/deletion and 100,000 random IP addresses lookup.
- The memory use is when there are 44,366 routes in the table.
- The simple backtracking method is used in MART and MART-CPE for deletion.
- SMART-TD uses the table default route method.

TABLE III
PERFORMANCE OF BSD RADIX, MART, MART-CPE, AND SMART

Table IV shows the distribution of the number of backtrack to find the second longest matching route in MART-CPE at deletion.

Table V shows the performance difference between two routing tables; the one (MAE-EAST) is the pure MAE-EAST routing table that has only 95 routes whose prefix length is longer than 24, the other (MAE-EAST+) is a table which has all the MAE-EAST routes plus 2,000 randomly created routes whose prefix length is longer than 24.

Table VI shows the performance difference between the following two cases:

1. 44,366 routes are added and deleted randomly
2. 44,366 routes are added according to the descending order of the prefix length and deleted according to the ascending order of the prefix length.

The IP addresses that have the same prefix length are

number of backtracks	number of routes	%
0	2,051	4.6
1	32,581	73.4
2 .. 9	7,371	16.6
10 .. 19	584	1.3
10 .. 39	778	1.8
40 .. 59	120	0.3
60 .. 99	535	1.2
100 .. 255	346	0.8

TABLE IV
BACKTRACKING NUMBER DISTRIBUTION (MART-CPE)

	level 1 array	level 2 array	search (μs/route)	add (μs/route)	delete (μs/route)
MAE-EAST	3,441	64	0.20	0.94	0.94
MAE-EAST+	5,299	2,064	0.20	1.13	1.13
diff.(%)	53.99	3,125	0.00	20.21	20.21

- MAE-EAST has 42,366 routes and 95 routes' prefix length is longer than 24
- MAE-EAST+ has 2,000 more routes whose prefix length is longer than 24 in addition to the MAE-EAST routing table

TABLE V
TABLE SIZE AND PERFORMANCE

added and deleted randomly. Search performance is measured with 100,000 random IP addresses lookup.

	search (μs/route)	add (μs/route)	delete (μs/route)
random	0.20	1.13	1.13
sorted	0.20	0.68	0.68

TABLE VI
UPDATE ORDER AND PERFORMANCE

The simulation result shows the following things:

1. SMART has 79.6% as fast as MART-CPE in deletion. It means that route deletion becomes very slow when a route flap happens even though 78% of the second longest matching routes are found with 0 or 1 backtracking. This is because:
(a) 'for' loop overhead is not negligible
(b) many memory accesses happen even though the percentage of long backtracking is low (about

22.0%).

This result shows that SMART is much better than MART-CPE when a route flap happens although SMART has few advantages against MART-CPE in the steady state.

2. SMART is more than 10 times in search, more than 3 times in addition and deletion as fast as BSD radix when the table default method is applied. On the other hand, SMART uses more than 3 times as much memory as BSD radix.
3. There is no difference in search performance among MART, SMART, and SMART with the table default route method. This result suggests that all the saved block default routes and the table default route are in the cache. It is quite possible because the Athlon CPU has 64K byte L1 data cache.
4. SMART improves route addition and deletion performance compared to MART. SMART is almost twice in route addition, more than 3 times in deletion as fast as the traditional MART.
5. The route addition and deletion performance improves 16% when the table default method is used. This result suggests that the cost of accessing the whole array affects the performance. Actually, the number of routes whose prefix length is 24 is large (21,971). These routes move to the level 2 arrays when more specific routes exist and it is necessary to access the whole array elements unless the table default route is applied.
6. Neither the number of routes nor the number of deep level arrays seriously affects the route update performance. The number of routes increased almost 54%, the number of level 2 array increased 32 times in Table V. However, the route update cost change is less than 17%. This is an expected result of the SMART route update algorithm.
7. When a route flap happens, the performance of both addition and deletion increases about 66% in the case routes are sorted in SMART. Modern routers usually have two routing tables. One is owned by the controller that handles routing protocols and the other is owned by the packet forwarder. The controller calculates routes and downloads them to the forwarder's routing table. The update performance of the forwarder increases about 66% when the controller sorts routes according to the prefix length and downloads them to the forwarder. It is inexpensive to sort routes according to only the prefix length.

The simulation result shows that the performance of SMART is pretty good for all 3 routing table operations. Furthermore, it is easy to implement the SMART search in hardware and make it pipelined. The search performance becomes single memory access speed in this case.

VIII. Conclusions

It is important to keep the route update cost low in order to quickly recover from route flaps. The number of IGP routes whose prefix length is longer than 24 is not negligible, either. The route update cost of simple MART is 32M and more routing table memory accesses in the worst case. We presented a new multi-array routing table called SMART (Smart Multi-Array Routing Table) whose route update cost is always less than 256 routing table memory accesses. SMART does not spread pointers to routes all over the routing table. Instead, each SMART array element has a block default route, which is the second longest-matching route of the associated array element. A block default route in an array does not become a block default route in any other arrays. In theory, the block default route affects the search performance, but it does not in reality. The SMART search cost becomes one memory access when it is implemented in hardware with pipelining. We showed the SMART search cost is practically the same as that of simple MART by simulation. We also showed that SMART is 10 times in search, more than 3 times in addition and deletion as fast as the BSD radix tree routing table by simulation.

Acknowledgments

We would like to thank Steve Reseigh, Paul Ziemba, Kevin Williams, and Donald Knuth for useful comments on a draft of this paper. We would like to thank MAYAN Networks for giving us an opportunity to write this paper. We would also thank Hiroki Nakano who helped to port the BSD radix code to a user program.

References

[1] Tony Bates, *Routing Table History*, http://www.employees.org:80/~tbates/cidr.plot.html
[2] V. Fuller, T. Li, J. Yu, and K. Varadhan, *Classless Inter-Domain Routing (CIDR)*, RFC1519, September 1993.
[3] Merit Networks, Inc., *Internet Routing Table Statistics*, http://www.merit.edu/ipma/routing_table/
[4] C. Labovitz, R. Malan, F. Jahanian, *Internet Routing Stability*, Proceedings of ACM SIGCOMM, Sept. 1997.
[5] Pankaj Gupta, Steven Lin, and Nick McKeown, *Routing Lookups in Hardware at Memory Access Speeds*, Proceedings of Infocom, April 1998.
[6] V. Srinivasan and George Varghese, *Faster IP Lookups using Controlled Prefix Expansion*, Proceedings of ACM Sigmetrics, Sep 98 and ACM TOCS 99.
[7] R. Hinden, M. O'Dell, S. Deering, *An IPv6 Aggregatable Global Unicast Address Format*, RFC2374, July 1998.
[8] FreeBSD 2.2.2, /usr/src/sys/net/radix.[ch]

Appendix 2

```
-- ###########################################################################
-- #
-- #      File: ip_route_engine.vhd
-- # Description: The following logic performs a longest match
-- #           route lookup using the structure and algorithm
-- #           defined by Yoichi Hariguchi of MAYAN Networks.
-- #
-- #           The logic will interface to an external control
-- #           entity in the form of an external hardware state
-- #           machine or a software driver. The logic also
-- #           interfaces with an external memory interface.
-- #           This memory interface takes an address and read
-- #           strobe and will return a 64bit vector read from
-- #           the given address.
-- #
-- # Assumptions: The routing tables must be aligned by size.
-- #           This allows the hardware to append the index value
-- #           to the base address of the routing table. This
-- #           will eliminates the need for large adders to be used
-- #           in the logic.
-- #
-- #           The memory interface will return a 64bit vector
-- #           that contains the 32bit DWORD read from the read
-- #           address base value in bits (31 downto 0). The 32bit
-- #           DWORD at address base + 1 will be returned in
-- #           bits (63 downto 32) of the returned vector.
-- #
-- ###########################################################################

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity ip_route_engine is port (

  -- state machine clock source.
  state_clk:        in  std_logic;

  -- state machine reset. asynchronous.
  state_rst:        in  std_logic;

  -- IP address to lookup. must be stable during entire lookup.
  lkup_ip_addr:     in  std_logic_vector(31 downto 0);

  -- Base address of the level 0 route table.
  route_table_base:  in  std_logic_vector(31 downto 0);

  -- route lookup start bit. set high to trigger route lookup. set
  -- bit low to clear the route engine after lookup is done.
  route_eng_run:   in  std_logic;
  -- memory ready input. set low by memory read logic when read cycle
  -- has completed and valid data is present on mem_read_value vector.
  mem_rdy_n:          in  std_logic;

  -- memory read result. data must stay stable until mem_rd_n has been
  -- set high by route engine.
  mem_read_value:    in  std_logic_vector(63 downto 0);

  -- memory read address.
  mem_read_pntr:     out std_logic_vector(31 downto 0);

  -- route lookup result vector. data is valid while route_eng_done
  -- bit is high.
  return_route:      out std_logic_vector(31 downto 0);

  -- route engine done indication. a '1' indicates that the route
  -- engine has completed its lookup and route vector can be read.
  route_eng_done:    out std_logic;

  -- memory read control. set low by route engine to read data from
  -- address contained in mem_read_pntr vector.
  mem_rd_n:          out std_logic

); end ip_route_engine;


architecture ip_route_engine_arch of ip_route_engine is

  -- current state variable supporting 8 states.
  signal  state:          std_logic_vector(2  downto 0);
  -- vector to store block default read from memory
  signal  block_default:  std_logic_vector(31 downto 0);
  -- vector to store route read from memory
  signal  read_route:     std_logic_vector(31 downto 0);
  -- signal to mux between memory read value and block default
  signal  return_bdef:    std_logic;

  -- State definitions. States are defined so the state machine
  -- can progress forward using a sequential counter.
  constant ST_IDLE:          std_logic_vector(2  downto 0) := "000";
  constant ST_TABLE_AB_READ: std_logic_vector(2  downto 0) := "001";
  constant ST_TABLE_C_ADDR:  std_logic_vector(2  downto 0) := "010";
  constant ST_TABLE_C_READ:  std_logic_vector(2  downto 0) := "011";
  constant ST_TABLE_D_ADDR:  std_logic_vector(2  downto 0) := "100";
  constant ST_TABLE_D_READ:  std_logic_vector(2  downto 0) := "101";
  constant ST_DONE:          std_logic_vector(2  downto 0) := "110";

begin


  -- Drive the output siganls
  drive_output: block
```

```
begin

  -- Set route engine done flag when state machine has completed the lookup.
  route_eng_done <= '1' when state = ST_DONE else '0';

  -- Set memory read flag when state machine is in any of the three memory
  -- read states.
  mem_rd_n <= '0' when (state = ST_TABLE_AB_READ) or (state = ST_TABLE_C_READ)
             or (state = ST_TABLE_D_READ) else '1';

        -- Mux between route entry and block default value based upon exit mode
        return_route(31 downto 0) <= read_route(31 downto 0) when return_bdef = '0'
          else block_default(31 downto 0);

end block drive_output;


-- Route lookup state machine.
state_tick: process ( state_clk, state_rst )
begin

  -- Set initial values on reset
  if state_rst = '0' then

          mem_read_pntr <= (others => '0');
                state        <= ST_IDLE;
                read_route    <= (others => '0');
                block_default <= (others => '0');
                return_bdef   <= '0';

        -- machine operates on rising edge of state clock
        elsif state_clk'event and state_clk = '1' then

    case (state) is

                  -- Idle state. Waiting for new IP address to be looked up.
                  when ST_IDLE =>

                          -- Wait for external entity to set the run bit, indicating that a new
                          -- IP address has been entered.
                          if route_eng_run = '1' then

                            -- Set the upper bits of the lookup address to the base address of
                            -- the level 0 routing table. Routing table must be aligned so
                                  -- bits (18 downto 0) of the base address are '0'.
                            mem_read_pntr(31 downto 19) <= route_table_base(31 downto 19);

                            -- Set bits 18 downto 11 to the A byte of the IP address to be looked up.
                            mem_read_pntr(18 downto 11) <= lkup_ip_addr(31 downto 24);

                            -- Set bits 10 downto 3 to the B byte of the IP address to be lookup up.
                            mem_read_pntr(10 downto 3 ) <= lkup_ip_addr(23 downto 16);
```

```
                    -- Go to the memory read state
                 state <= state + 1;

                end if;


      -- Common logic is used for all three memory read states
               when ST_TABLE_AB_READ | ST_TABLE_C_READ |
ST_TABLE_D_READ =>

       -- Wait for the memory read logic to indicate the read cycle has completed.
                if mem_rdy_n = '0' then

                 -- Update block default value if it contains valid data
                    if mem_read_value(31 downto 2) /=
"000000000000000000000000000000" then
                      block_default(31 downto 2) <= mem_read_value(31 downto 2);
                          block_default( 1 downto 0) <= (others => '0');
                 end if;

                    -- Update the read route value. This is the second DWORD read
                    -- from memory.
                    read_route(31 downto 0) <= mem_read_value(63 downto 32);

                    -- Update the mem_read_pointer value. This is the second DWORD
read

                    -- from memory. The next level table must be aligned so that
                    -- bits (10 downto 0) of the base address are '0'.
                    mem_read_pntr(31 downto 11) <= mem_read_value(63 downto
43);

                 -- If the lower two bits of block default are "01" then the pointer block
                    -- contains a pointer to the route entry for this IP address. The
routing

                    -- engine returns the route value read.
                    if ( mem_read_value(1 downto 0) = "01" ) then
                   return_bdef <= '0';
                     state <= ST_DONE;

                 -- If the lower two bits of block default are "10" then the pointer block
                    -- contains a pointer to the next level routing table. The routing
engine

                    -- continues to the next table using the IP address as an index.
                    elsif ( mem_read_value(1 downto 0) = "10" ) then
                          state <= state + 1;

    -- If the lower two bits of block default are "00" the pointer block does
                    -- not contain a valid pointer. Return the block default.
    else
                      return_bdef <= '1';
                          state <= ST_DONE;
                    end if;
```

```
        end if;

      -- Generate address for TABLE 1 (byte C of IP) lookup.
                      when ST_TABLE_C_ADDR =>

                        -- Set bits 10 downto 3 of the memory pointer to the C byte of the
                        -- IP address to be lookup up. Table must be aligned so that
                        -- bits (10 downto 0) of the base address are '0'.
                        mem_read_pntr(10 downto 3 ) <= lkup_ip_addr(15 downto 8 );

                        -- Go to the memory read state
                        state <= state + 1;

                      -- Generate address for TABLE 2 (byte D of IP) lookup.
                      when ST_TABLE_D_ADDR =>

                        -- Set bits 10 downto 3 of the memory pointer to the D byte of the
                        -- IP address to be lookup up. Table must be aligned so that
                        -- bits (10 downto 0) of the base address are '0'.
                        mem_read_pntr(10 downto 3 ) <= lkup_ip_addr(7 downto 0 );

                        -- Go to the memory read state
                        state <= state + 1;

                      when ST_DONE =>

                        -- Wait here until external entity has cleared the run bit. This indicates that
                        -- the route has been read and the engine is free to lookup a new route.
                        -- Clear the values stores in block default and read route.
                        if route_eng_run = '0' then
                          state        <= ST_IDLE;
                                read_route    <= (others => '0');
                                block_default <= (others => '0');
                        end if;

                      when OTHERS =>
                        state <= ST_IDLE;

    end case;
        end if;
  end process state_tick;
end ip_route_engine_arch;
```